SAFE REINFORCEMENT LEARNING: AN OVERVIEW, A HYBRID SYSTEMS PERSPECTIVE, AND A CASE STUDY

BY

MATHEW POTOK

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

Adviser:

Professor Sayan Mitra

ABSTRACT

Reinforcement learning (RL) is a general method for agents to learn optimal control policies through exploration and experience. Due to its generality, RL can generate novel policies that may not be easily expressed with rules-based strategies or traditional control techniques. Over the years since its inception, RL has been able to solve increasingly more challenging control problems, from GridWorld to Go. Despite these impressive results, the successes of RL have been predominantly limited to systems with discrete environments and agents, particularly video and board games.

A key barrier to using RL in safety-critical cyber-physical system applications is not only transferring these results to continuous domains but also ensuring that a notion of 'safety' is upheld during the learning process. This thesis highlights some of the recent contributions in safe learning and presents a framework, FoRShield, for learning safe policies of a control system with nonlinear dynamics. The framework develops a generic hybrid systems model for online RL. The model is used to formalize a shield that can filter unsafe action choices and proved feedback to the underlying RL system.

The thesis presents a concrete approach for computing the shield utilizing existing reachability analysis tools. The feasibility of this approach is illustrated against a case study with a quadcopter that uses RL to discover a safe and optimal plan for a dynamic fire-fighting task. The approach is realized as an open-source framework, FoRShield. The framework is implemented in Python in a modular fashion to allow for testing of a variety of algorithms. Our particular implementation utilizes the Actor-Critic algorithm to learn policies. The experiments show that interesting fire-fighting strategies can be safely learned for a discrete environment with 2^{32} states and a 9-dimensional plant model using a standard laptop computer.

To my wife and parents, for their love and support.

ACKNOWLEDGMENTS

I would like to express my special appreciation and thanks to my adviser, Professor Sayan Mitra, for advising me through both my undergraduate and graduate research. Working with him through all these years has provided many exciting opportunities and has opened countless more. His guidance and patience have helped me tremendously to grow as student and researcher.

Many thanks to all the members of our research group past and present: Parasara Sridhar Duggirala, Zhenqi Huang, Ritwika Ghosh, Chuchu Fan, Yixiao Lin, Bolun Qi, Hussein Sibai, and Nicole Chan. There were many discussions between us, friendly and serious, that livened up the atmosphere in the office and provided much food for thought. You all helped make the office in CSL feel like my second home.

I would like to thank other collaborators with whom I have worked throughout my university journey, particularly: Professor Mahesh Viswanathan and Suket Karanwat with Rational CyPhy; Professor Sibin Mohan and Chien-Ying Chen with SDCWorks; and Professor Sanjay Patel and Professor Yuting Chen for the special privileges as an undergraduate TA.

I am indebted to my parents for their unconditional love and dedication. They have provided me with a fantastic opportunity, one that was not readily available to them. Extraordinary thanks to my wife, Ge Yu, for her love and support. She brightened many of my days and provided much-needed encouragement.

Finally, I would like to acknowledge the support of Boeing and the National Science Foundation for supporting this and other works.

CONTENTS

Chapter 1

INTRODUCTION

1.1 Motivation

The field of AI, particularly machine learning (ML) and reinforcement learning (RL), has enjoyed exponential investment and growth in the past few years. We now interact with numerous AI algorithms every day through email spam filters, recommendations systems, news curation applications, etc. Slowly, these AI algorithms are encroaching into the territory of cyber-physical systems (CPS) products such as critical infrastructures, robotics, supply chain, aviation, and most notably self-driving cars.

Almost all car manufacturers these days along with numerous other large companies and start-ups are either developing autonomous vehicle capabilities in-house or creating partnerships to gain access to such capabilities. A subset of these companies have developed viable prototypes and have started testing them out in the real-world. Testing autonomous vehicle prototypes outside of simulation environments and in the real-world offers a wealth of information, but there are enormous consequences when these autonomous systems fail. Two notable accidents [1, 2] resulted in fatalities and even partially halted any further testing in the real-world [3]. Although the use of AI black boxes in fully autonomous vehicles and cyber-physical systems in general may provide many potential benefits to humanity, the challenges of safety and security are yet to be met.

As an increasing number of cyber-physical systems begin to either incorporate or replace existing modules with AI modules, methodologies to formally reason about these modules become crucial. A key challenge in applying AI, whether it be ML or RL, to CPS products lies in specifying a degree of correctness for the system and verifying that this correctness holds. The processes utilized for developing, training, and updating AI modules are rad-

ically different from those used for traditional software and cyber-physical systems. Not surprisingly, the traditional tool-chains and methodologies do not suit AI algorithms and modules particularly well.

At the traditional methodologies end of the spectrum, the approaches rely heavily on formulating well-defined mathematical models to reason about various properties of systems. These carefully crafted mathematical models are subjected to model checking and to theorem provers to mathematically guarantee that certain properties have been met. Even though such rigorous approaches yield definite results, creating mathematical formulations may not be a straightforward process.

The other end of the spectrum relies on the 'brute force' approach where a system is exhaustively tested against all possible cases. This approach will also certify a truth, but is severely limited by the size of the set of cases. Current verification strategies for AI modules seem to have opted for this approach. As a case in point, several autonomous vehicle manufacturers announced the total miles logged by their vehicles as a proof of safety of their autonomous systems [4, 5]. Although total logged miles may seem a good indicator of safety through the lenses of arguments such as PAC learning, "when a metric is used as a target, it ceases to be a good metric" [6]. Such simple metrics make it easy to judge the approximate safety of AI modules in general, but fail to quantify the true safety of the system. Certain companies like Righthook.io make claims such as "11,000,000,000 miles are needed to reach statistical confidence an autonomous vehicle has exceeded human driving by 20%" [7]. Yet, considering the numbers logged by manufacturers and the performance of their autonomous vehicles, these claims of super-human driving ability are comically meaningless. The brute force approach provides a quick way to analyze AI modules but in the end, "program testing can be used very effectively to show the presence of bugs but never to show their absence" [8].

Currently, the technical challenges of verifying AI modules are being address by two broad research efforts. The first effort targets analysis and verification of individual AI modules. For example, neural network (NN) image classifiers show promise on real-world data sets [9] but they are known to be fundamentally fragile to adversarial examples [10]. There is a growing body of research on testing networks and hardening them against adversarial example attacks [10].

The second line of research addresses the problem of end-to-end testing and verification systems that use AI modules. In [11], the authors make a case for using the system-level specifications for guiding the search for adversarial examples in ML. A fault injection-based resiliency assessment system for autonomous vehicles is presented in [12]. Along similar lines, [13] presents a fault injection framework to assess the resilience of openpilot [14], under different environmental conditions and sensor faults.

1.2 Thesis Overview

The major contributions of this thesis fall into the latter category of addressing the holistic verification of AI modules. Our work considers system-level safety verification of a CPS in which a key decision making module is developed (trained) with RL. We present a framework, FoRShield, that uses the Actor-Critic RL algorithm to simultaneously verify the actions of an agent with continuous dynamics in a discrete environment. The framework relies on a generic hybrid model of a control system that is defined by a learned high-level planner through RL and an underlying low-level controller for achieving the individual decisions suggested by the planner. The hybrid formalism enables us to clearly specify the semantics of the overall system and helps identify the information flow through the different modules involved.

Our problem setting involves a hierarchical control system: a low-level controller drives the system to near-term *goals* or waypoints, and a high-level controller or *planner* chooses the sequence of goals to accomplish a longer term *mission*. We consider situations, where existing control design techniques (for example, PID, LQR, MPC, etc.) are adequate for designing the low-level controller, but the design for the planner is challenging. This is a common scenario where the planner has to optimize for multiple objectives to accomplish the overall mission and simultaneously adhere to the constraints imposed by the low-level planner.

The hybrid model formulation is used to define a shield, a protective structure that can filter unsafe action choices both online throughout the learning process and offline during execution. Additionally, the shield provides feedback to the RL algorithm by penalizing it for selecting unsafe actions with a very negative reward. If there exist safe paths in the environment, then the

system is guaranteed to converge to a policy with no unsafe actions.

The continuous agent and the environment together form the MDP of the RL problem. The dynamics of the continuous agent are used by the shield to verify that proposed actions are safe with respect to the environment. An overview of the complete framework is presented in Figure 1.1. Additionally, we present a method to reuse reachability analysis computations for agents with translation invariant dynamics.

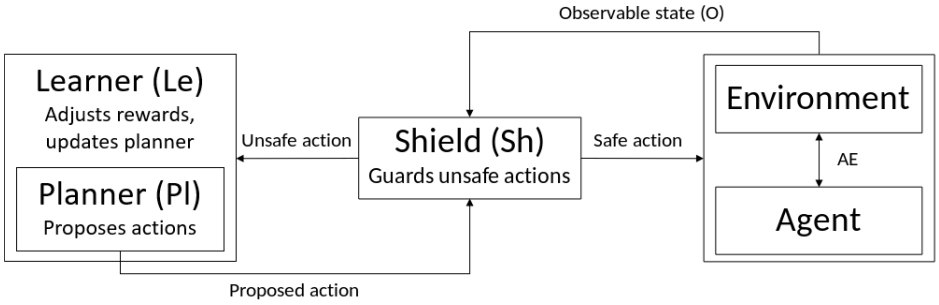

Figure 1.1: Illustration of modules in an RL-based control system and the information flow between them

We evaluate FoRShield against a case study of autonomous drone fire fighting. The drone in the case study is defined by a 9-dimensional continuous dynamics model derived from [15]. In the case study, the objective of the drone is to extinguish a fire within a grid environment as efficiently as possible while minimizing the spread of fire and the chances of collision with obstacles. To this end, the high-level planner has to decide the location and direction for approaching the fire (which may change over time); it has to choose sources for collecting water and schedule collection; it has to avoid obstacles.

Finally, we compare our shield against two baseline methods and demonstrate that our framework is capable of learning safe and near optimal fire-fighting policies for an agent with continuous nonlinear dynamics. In the various scenarios we tested, the policies learned by the agents through our framework were guaranteed to be safe and prevented the agent from entering any unsafe states

1.3 Contributions

The contributions of this thesis are:

1. An up-to-date literature review of research in the areas related to safe AI and formal verification in the AI community
2. A framework that formalizes RL-based control systems as hybrid systems
3. A proposal for shields for non-linear plant models and deep-learning-based RL algorithms in conjunction with existing hybrid verification techniques
4. An experimental feasibility test of our approach on an interesting case study
5. An implementation of the framework available on request

1.4 Organization

The remainder of the thesis is organized as follows. A survey of recent contributions in the area of safe RL is presented in Chapter 2. Chapter 3 introduces a framework for modeling systems as a hybrid automaton. Chapter 4 presents the fire-fighting case study. Our implementation of the shield using reachability analysis along with the FoRShield framework is introduced in Chapter 5. The experimental results and implementation details are outlined in Chapter 6. Lastly, Chapter 7 concludes the thesis and suggests possible future lines of research.

Chapter 2

RELATED WORK ON SAFE RL

In this chapter, we present an short overview of safe RL works related to our framework FoRShield. Section 2.1 provides definitions of concepts that will be used in the literature survey and in later chapters. Section 2.2 recounts the recent history of AI and how it evolved into its current state. Sections 2.3 and 2.4 each cover a unique approach to safe RL and describe several works based on that approach.

2.1 Background

A probability distribution over a finite set X is a function $\mu : X \to [0, 1] \subseteq \mathbb{R}$ with $\sum_{x \in X} \mu(x) = \mu(X) = 1$. The set of all distributions on X is denoted by $Distr(X)$.

A Markov decision process (MDP) is a tuple $\mathcal{M} = \langle \mathcal{S}, s_I, \mathcal{A}, \mathcal{P}, \mathcal{R} \rangle$ where

1. $s \in \mathcal{S}$ is a finite set of states
2. s_I is the set of initial states
3. $a \in \mathcal{A}$ is the finite set of states
4. $\mathcal{P} : \mathcal{S} \times \mathcal{A} \to Distr(S)$ is a state transition probability function
5. $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is an immediate reward function.

Reinforcement learning (RL) is a class of algorithms that allows agents in a particular and possibly unknown environment to learn optimal policies through trial and error by maximizing (or minimizing) a predefined reward (or cost) function. Environments in RL are typically formulated as MDPs. The agent and the environment interact with each other in a cyclical fashion. At a particular time t, the agent observes the state of the environment $s_t \in \mathcal{S}$

and chooses an action $a_t \in \mathcal{A}$. The environment evolves to state $s_{t+1} \in \mathcal{S}$ with probability $\mathcal{P}(s_{t+1}|s_t, a_t)$ and returns an immediate reward $\mathcal{R}(s_t, a_t, s_{t+1})$. The agent utilizes the immediate returned reward to compute the cumulative total return $R = \sum_{t=0}^{T} \gamma r_t$ where $\gamma \in [0, 1]$ is a discount factor used to control the influence of future immediate rewards. By maximizing the expectation of the total return, $\max_{\pi \in \Pi} E_\pi[R]$, the agent is able to learn an optimal policy π^* from the set of all policies $\pi \in \Pi : \mathcal{S} \to \mathcal{A}$.

The most basic RL algorithms, policy iteration and value iteration, utilize tables to store a value for either each state or each state-action pair defined by an MDP. These algorithms iteratively recompute each of the values in the table according to an update function until all the values converge to some constant. However, for certain problems whose MDPs contain extremely large state-spaces, it may be prohibitive or infeasible to create enormous tables. An alternative to using tables is to use function approximators as estimators for the values of either states or state-action pairs [16]. Function approximators such as neural networks can achieve performance similar to that of tabular methods while using significantly less memory. The class of RL algorithms that utilize neural function approximators is called deep RL (DRL).

Neural networks (NNs) are universal function approximators that are formed by creating a network of consecutively layered neurons. Given a function $y = f(x)$, NNs are able to model $y = g(x)$ such that $g(x) \approx f(x)$. Input is fed into the network via the input layer. The input layer passes on the input to the hidden layer(s) where it is processed by a system of weighted connections. The final hidden layer is linked to an output layer which yields the final output. Each layer in the network yields a vector of values, i.e. $y_i = \sigma_i(W_i x_i + b_i)$ where i refers to the ith layer in the network, x_i is the output from the previous layer, W_i are the weights of the neurons in the layer, b_i is the bias, and σ_i is the activation function. Every neuron in a layer contains an activation function whose primary function is make an NN nonlinear. There is a wide variety of activation functions to chose from, and different layers in the network may use different activation functions.

The backpropagation algorithm is a gradient descent method used to train a network of neurons through the use of gradients [17]. A loss function is defined at the output layer that compares the actual output of the network with an actual output. As the results of the loss function are distributed

backwards throughout the network's layers, the weights on the connections between neurons of neighboring layers are updated accordingly.

Safe RL can generally be defined as a "process of learning policies that maximize the expectation of the return in problems in which it is important to ensure reasonable system performance and/or respect safety constraints during the learning and/or deployment processes" [18]. It is a similar approach to RL except it considers additional safety constraints and strives to find optimal policies that satisfy these constraints. Policies learned through safe RL should be verifiable and certifiable for critical systems such as CPS.

2.2 Recent History of AI

Starting in the early 2000s, the AI community underwent a resurgence as some important hurdles were overcome and more powerful hardware became readily available. One such major hurdle was the vanishing gradient descent problem [19] which prevents the earlier layers of deeper neural networks from receiving the back-propagated error and updating the weights of these layers in a meaningful way. It was discovered [20, 21] that this was not a fundamental problem with neural networks; rather it was an issue with applying gradient based learning methods to some classes of activation functions such as *sigmoid* and *tanh*. Moreover, the creation of extremely large data sets such as ImageNet [22] allowed researchers to focus on optimization problems rather than tedious data collection, and provided a baseline against which everyone could compare their results. Competitions sponsored by prominent conferences around these data sets promoted healthy competition among researches and brought about noteworthy architectures like AlexNet [23], VGG [24], GoogleNet [25], and ResNet [26], each contributing new techniques to the proverbial AI table.

These breakthroughs in the machine learning community, particularly regarding image tasks, enabled impressive results in other AI communities such as reinforcement learning. The application of deep neural networks as function approximators coupled with asynchronous RL algorithms and significant compute power allowed researchers to solve a suite of Atari games with scores better than those of human counterparts [27]. These new methodologies were applied to increasingly complicated games involving much larger state spaces

and requiring strategies spanning extended time horizons such as Go [28], DotA [29], StarCraft [30]. The learned agents in the first two listed game environments were able to compete and even defeat the top-ranked professional players. As a real-time strategy game, StarCraft can be considered one of the hardest environments in the video game domain. There has been some initial progress creating modest agents, but new advancements will be required to conquer this game along with a few other remaining bastions.

Even though impressive results have been achieved year after year in the AI communities, most of the previously mentioned approaches have forgone upholding safety constraints. Notably, neural networks have been shown to be susceptible to adversarial attacks. These attacks modify the input in imperceptible ways such that a human cannot distinguish between the original and modified versions. However, deep learning algorithms will incorrectly classify the modified examples with high probabilities [31]. Similarly, this oversight is evident in reinforcement learning where agents are repeatedly exposed to dangerous conditions during the training process. Such exposure may not always be feasible to agents situated outside sand-boxed game environments where mistakes are inconsequential. Furthermore, there are no guarantees that agents trained in this fashion will not violate some constraints. As more AI works are integrated into safety-critical systems, it becomes a necessity that there exist methods for verifying AI black boxes and ensuring that off-the-shelf AI components meet predefined specifications.

Several comprehensive surveys detail the recent literature for safe AI [18, 32, 33]. This chapter seeks to complement these works with a brief survey of safe reinforcement learning literature, particularly detailing works most similar to our approach described in later chapters.

2.3 Shield Learning

One approach to safe RL is to limit the actions of an agent to only a safe subset and utilize RL to learn a safe policy. A shield is such a construct that determines which set of actions is safe. The works below present several approaches to synthesizing shields.

In [34], the authors propose an approach for learning optimal policies while enforcing safety properties expressed in temporal logic, particularly linear

temporal logic (LTL). The work introduces shield learning, a learning strategy augmented by a synthesized reactive system that prevents an agent from executing unsafe actions in an environment. The shield is inserted into the traditional RL setting loop in one of two suggested positions. In the first position, the shield reduces the complete action set at each state to those actions that are definitely safe. In the second position, the shield monitors an agent's selected actions and corrects them if necessary. To synthesize a shield, the product is taken between a provided safety specification formulated in LTL and an abstraction of the environment to construct a notion of a game that is similar to that of an MDP. This construction is then used to learn safe optimal policies. One downside of this approach is that it relies on an omniscient model to determine the set of unsafe states at each state. Such an all-knowing model may not be readily available or even feasible.

An alternative approach to shield construction is proposed by the authors of [35] who consider the notion of a scheduler to limit the action set of an agent. The objective is to find a optimal scheduler that satisfies the specified safety constraints and that simultaneously minimizes (or maximizes) the expected cost (reward) to reach a goal state. Specifications are encoded as temporal logic constraints in the form of probabilistic computation tree login (PCTL) properties and are a combination of a reachability property, $P_{\leq\lambda}(\diamond T)$, and an expected cost property, $E_{\leq\kappa}(\diamond G)$. The reachability property upper-bounds the probability of eventually reaching a set of states T to be $\leq \lambda$. Similarly, the expected cost property upper-bounds the expected cost of eventually reaching the set of states G. Given these safety properties, the paper describes an algorithm for finding a safe optimal policy as follows. First, the environment MDP in conjunction with the reachability property is reformulated into an SMT encoding. Next, this SMT encoding is passed into an SMT solver to synthesize a safe permissive scheduler which is actually a subset of the original environment MDP that adheres to the reachability property. Then, traditional RL algorithms are run on this sub-MDP to discover safe policies. Finally, the resultant expected cost is compared against the specified expected cost to see whether an optimal policy has been discovered. If not, another SMT encoding is formulated that excludes the previous schedulers. Although this approach can output safe policies, it essentially boils down to a brute-force method that will try a copious number of schedulers. The authors briefly acknowledge this incremental nature

of their algorithm and suggest possible improvements.

The above authors expand on their work in [36] and apply their work to the more complicated problem of PAC-man. The major differences between this work and the previous work is synthesis of the shield. A similar reachability property is considered as before, $P_{\geq\lambda}(\diamond T)$, but it lower-bounds the probability of eventually reaching the set of states T. To synthesize a shield, an action-valuation is evaluated for every action in every state to determine the maximal probabilities of satisfying the safety property. Such a formulation may not be tractable, so the authors limit the probability computation to a finite horizon. Given the complete set of probabilities, a subset of the environment MDP can be formulated that satisfies the reachability property. The authors implement their algorithm and experiment against the PAC-man environment. Even though the implemented approach was unable to guarantee complete safety, it was able to decrease training times by preventing the agent from dying as often as it would without the presence of a shield.

2.4 Sketching

Rather than synthesizing shields that can reduce the complete set of actions to a subset of safe actions, sketching approaches can synthesize a policy directly in a human-readable output. The outputs generated through sketching can be verified with established traditional verification methodologies.

Such an approach is considered by the authors of [37] who propose VIPER, an algorithm that combines ideas from model compression and imitation learning to learn decision tree policies through policy extraction. The benefits of decisions tree are that they are nonparametric so they are capable of representing complex policies comparable to those of deep neural networks (DNNs) and they are efficiently verifiable through existing methodologies. The extraction technique utilizes imitation learning of high-performing DNN policies to guide the training, particularly DAGGER [38]. The reason for generating decision tree policies through supervised learning rather than direct learning is that DNNs are better regularized and easier to train, especially with gradient descent methods. To leverage both the action and cumulative reward output by a DNN, the authors extend DAGGER to Q-DAGGER and additionally modify Q-DAGGER to output decision tree policies in VIPER.

The output policies can then be verified with methodologies such as sum-of-squares (SOS) or satisfiability (SAT) optimization.

Another sketching approach is outlined in [39]. The authors propose a learning framework called programmatically interpretable reinforcement learning (PIRL). This framework is parameterized by a high-level programming language for policies whose semantics consist of atoms and sequences of atoms. The suggested language may be represented compactly and canonically to facilitate searching over the space of programs. To find an optimal program, the authors first utilize deep reinforcement learning to find a policy that will serve as an approximation for searching for a programmatic policy. The algorithm then uses an imitation learning approach to iteratively searches through a set of program templates with the use of Bayesian optimization to find optimal parameters for the templates and generates a pool of template candidates. These candidates are simulated against the environment and the total rewards are gathered. This procedure will iterate until none of the candidates return a higher reward than the current maximum. The authors test their framework against a few classic control problems in RL and TORCS, a driving simulator, and verify that the generated policies adhere to their defined safety specifications. A domain expert may be required to develop the programming language constructs from which the generated policies will be formulated and to guide the learning process.

The author's of [40] build upon the two works introduced above by wrapping them inside a overarching framework, mixed optimization scheme for reinforcement learning (MORL), for synthesizing policies with repairing capabilities. To synthesize an optimal policy, the framework first starts with an initial policy π_0 and uses synthesis techniques similar those discussed previously to learn a symbolic representation of a learned policy as a program P_0. With this programmatic representation, it is possible to perform program repair either manually by a human expert or automatically using safety specification constraints to remove or add certain behaviors. The repaired program P_0' is then transferred into an improved policy π_0' that is further improved with standard gradient descent. Once a minima is reached, the policy π_0' becomes π_0 for the next iteration if necessary. The complete procedure can be visualized as $\pi_t \rightarrow P_t \rightarrow P_t' \rightarrow \pi_t' \rightarrow \pi_{t+1}$ where t is the loop counter. The MORL framework improves on the above works by utilizing repetitive optimization of programs to converge to optimal solution.

However, it may be vulnerable to converging to less optimal solutions if the repaired programs are worse then the original generated program.

Chapter 3

LEARNING-BASED CONTROL SYSTEMS MODEL

We consider a system consisting of an *agent* acting on an *environment* based on the directions of a (high-level) *planner*. The planner's mission is to drive the environment to a target state by setting intermediate *goals* for the agent. The planner will be learned using RL. The agent has a (low-level) *controller* that can achieve any of these intermediate goals chosen by the planner. In the example fire-fighting application we discuss in Chapter 4, the agent is a drone that moves in a 3D world to collect and deploy water; the environment is a 2D map of a spreading wildfire, water sources, and possibly other obstacles available from sensors. The planner's mission is to douse the fire, and to this end it sets goals for the drone to move and collect and deploy water.

In order to learn a planner online, we will use a *learner* and a *shield*. The shield forward analyzes a goal proposed by the planner to evaluate whether it is safe to pursue from the current state of the agent. The learner uses this output from the shield and the state to update the planner.

In the rest of this chapter, we develop a complete formal model of the hybrid system parameterized by all these components—agent, environment, shield, learner, and planner. Each planner goal is considered as a separate mode in the hybrid automaton. In every mode, the agent state evolves according to a differential equation (determined by the low-level controller). There are three types of discrete transitions. First, the environment state is updated periodically independent of the agent (env transitions). Second, when the agent achieves a goal, the environment, the agent, the planner, and the goal are updated (pln transitions). If the goal is not safe, immediately, the planner gets updated and a new safe goal is set by the new planner (repln transition). This last type of transition may occur several times before a safe goal is found. Figure 1.1 shows the key pieces of the hybrid automaton.

3.1 Agent, Environment, and Planner

Formally, let $S_a = \mathbb{R}^n$ be the continuous *state space of the agent* for some $n \in \mathbb{N}$, and let O_a be the set of observable states of the agent. Let $\mathsf{Obs} : S_a \to O_a$ and $\mathsf{Obs}^{-1} : O_a \to 2^{S_a}$ be the (invertible) state observation map and its inverse.

Let S_e be the discrete *state space of the environment* (as sensed by the agent). We define $O = O_a \times S_e$ to be the set of observable states.

Let G be the finite set of modes or goals that can be chosen by the planner.

A *planner* is a function $\mathsf{Pl} : O \times G \to G$ that maps an observed state of the agent, an observed state of the environment (as seen by the agent), and a (current) goal to a new goal. We imagine this to be a complicated function that is going to be learned using RL with runtime state observations. We denote the set of all possible planners by $\mathcal{P}$.

As we shall see in Section 3.4, the state space of the hybrid automaton describing the overall system will be $Q = S_a \times S_e \times G_\perp \times \mathcal{P}$, where $G_\perp = G \cup \{\perp\}$. The $\perp$ value will indicate that a goal has not been decided by the planner. For a state $q \in Q$, we denote the first, second, third, and fourth elements by $q.S_a, q.S_e, q.G, q.\mathcal{P}$, respectively. We call q a system state. A (bounded) trajectory ξ for Q is a map from an interval of $[0, T]$ to Q, for some $T \geq 0$. We denote its domain by $\xi.dom$, its first state $\xi(0)$ by $\xi.fstate$, and its last state $\xi(T)$ by $\xi.lstate$. A trajectory ξ with $\xi.dom = [0, 0]$ is called a point trajectory.

3.2 Environment and Agent Updates

The environment of the agent evolves dynamically independent of the agent, but its effects are seen by the agent through periodically updating sensors. These updates are captured by a (possibly nondeterministic) function $\mathsf{En} : S_e \to 2^{S_e}$, which we call the *environment update function*.

A *guard* $\mathsf{Grd} : G \to 2^{S_a}$ maps each goal $g \in G$ to a set of agent states that correspond to accomplishing that goal. An *agent-environment joint update function* $\mathsf{AE} : S_a \times S_e \times G \to S_a \times S_e$ gives the (joint) new state of the environment and the agent after the agent accomplishes a goal. As the name suggests, only this function captures the interactions of the agent

and the environment. For example, when the fire-fighting drone arrives at a fire location and deploys water, this function updates the water-level in the drone and also updates the environment with a new (lower) level of fire.

3.3 Shield and Learner

The system-level requirements are specified by two sets: a *mission target* set $S_e^* \subseteq S_e$ and a set of unsafe states $U \subseteq S_a \times S_e$. U' is the lifting of U to Q. That is, $q \in U'$ iff $q.(S_a \times S_e) \in U$. We will write U instead of U' when the type is clear from context.

With a fixed unsafe set U, a *shield* takes as input a current (observed) state of the agent, state of the environment, and a candidate planned goal, and decides whether or not this goal is a safe one to pursue for the agent. Thus, the shield is a function of the type, $\mathsf{Sh} : O \times G \to \{safe, unsafe\}$.

A *learner* updates the current planner with a new planner based on the current observed state (of agent and environment) and the current goal, with the objective of achieving mission target while preserving safety. Thus, it is a function of the type $\mathsf{Le} : \mathcal{P} \times O \times G_\perp \to \mathcal{P}$.

3.4 System-level Hybrid Automaton

Given all of the above sets and maps, now we are ready to define the hybrid automaton that describes the behavior of the overall system.

Definition 1. *Given all the functions introduced in the previous section, the hybrid automaton describing the system is a 4-tuple* $\mathcal{H} = \langle Q, \Theta, \mathcal{D}, \mathcal{T} \rangle$ *that is defined as follows:*

(i) $Q = S_a \times S_e \times G_\perp \times \mathcal{P}$ is the state space of the automaton.

(ii) $\Theta \subseteq Q$ is the set of initial states.

(iii) $\mathcal{T}$ is a set of trajectories for Q. Along any trajectory $\xi \in \mathcal{T}$, the environment state, the planner, and the goal remain constant and only the agent state evolves continuously.

(iv) $\mathcal{D} \subseteq Q \times \{\mathsf{env}, \mathsf{pln}, \mathsf{repln}\} \times Q$ is the set of discrete transitions. A $(q, \ell, q') \in \mathcal{D}$ iff one of the following conditions hold:

(a) (environment transition) If $\ell = \mathsf{env}$ and time-elapsed in q[1] since the last environment transition is τ then a transition is enabled. The post-state of the transition updates $q'.S_e \in \mathsf{En}(q.S_e)$, and all other components, remain unchanged, i.e., $q'.S_a = q.S_a$, $q'.G = q.G$, and $q'.\mathsf{PI} = q.\mathsf{PI}$.

(b) (planner transition) If $\ell = \mathsf{pln}$, $q.G \neq \perp$ and $q.S_a \in \mathsf{Grd}(q.G)$, a transition is enabled. The agent and environment states get updated: $(q'.S_a, q'.S_e) = \mathsf{AE}(q.S_a, q.S_e, q.G)$. Moreover, the planner gets updated as:

$$q'.\mathsf{PI} = \mathsf{Le}(q.\mathsf{PI}, \mathsf{Obs}(q.S_a), q.S_e, q.G).$$

Let $o'_a = \mathsf{Obs}(q'.S_a)$ and $g = q'.\mathsf{PI}(o'_a, q'.S_e, q.G)$. If $\mathsf{Sh}(o'_a, q'.S_e, g) = \mathit{safe}$, then $q'.G = g$. Otherwise $q'.G = \perp$.

(c) (re-plan transition) If $\ell = \mathsf{repln}$ and $q.G = \perp$, a transition is enabled. The agent and environment states stay constant. Let $o_a = \mathsf{Obs}(q.S_a)$, then the planner gets updated as:

$$q'.\mathsf{PI} = \mathsf{Le}(q.\mathsf{PI}, o_a, q.S_e, \perp),$$

Let $g = q'.\mathsf{PI}(o_a, q.S_e, \perp)$. If $\mathsf{Sh}(o_a, q.S_e, g) = \mathit{safe}$, the goal is updated as $q'.G = g$. Otherwise, the goal is kept $\perp$.

A few remarks on Definition 1. First, discrete state space and periodic update model for the environment is an abstraction of sensing and perception modules of the agent. These modules collect and fuse information, possibly from distributed sensors, and present a view of the environment to the decision-making modules in the agent, namely the planner.

Second, for any trajectory $\xi \in \mathcal{T}$, the goal, planner, and environment remain constant. That is, at any time $t \in \mathit{dom}(\xi)$, $\xi(t).G = \xi(0).G$, $\xi(t).\mathsf{PI} = \xi(0).\mathsf{PI}$, $\xi(t).S_e = \xi(0).S_e$, and $\xi(t).S_a$ changes continuously with time. In special cases, we may assume that the $\xi(t).S_a$ is a solution of a known differential equation. That is, for a given starting state $q_0 \in Q$ with $q_0.G \neq \perp$,

[1]This is checked with a *timer* variable that is omitted in the definition.

the function $\xi(\cdot).S_a$ satisfies the differential equation:

$$\frac{d}{dt}(\xi(t).S_a) = f_{q_0.G}(\xi(t).S_a), \tag{3.1}$$

with $\xi(0).S_a = q_0.S_a$. The dynamics of the low-level controller may depend on the chosen goal, and hence, the right-hand side of the ODE (3.1) depends on $q_0.G$.

Definition 2. *A hybrid automaton of Definition 1 is said to be* goal-tracking *if for each $g \in G$, for every trajectory $\xi \in \mathcal{T}$ with $\xi(0).G = g$, there exists a time T^* such that $\xi(T^*).S_a \in \mathsf{Grd}(g)$ and for all $t > T^*$, $\xi(t).S_a \in \mathsf{Grd}(g)$.*

The goal tracking property ensures that once a goal $g \in G$ is set by the planner, the low-level controller of the agent drives the agent state s_a to the corresponding set $\mathsf{Grd}(g)$.

The final remark is about the transitions. There are three types of discrete transitions in the hybrid automaton of Definition 1. All of these types of transitions are mutually exclusive and *urgent*; that is, at most one of them is enabled in any given state, and they do indeed occur as soon as they become enabled [41].

First, environment transitions (env) are enabled every $\tau > 0$ time, and when the transition occurs, the discrete state of the environment s_e is updated, possibly nondeterministically, according to the environment update function En.

Second, planner transitions (pln) are enabled at a state q iff $q.G \neq \perp$ and $q.S_a \in \mathsf{Grd}(q.G)$. The state of the environment and the agent get updated, possibly nondeterministically, according to the joint update function AE. Furthermore, the learner Le updates the planner based on the current observed state and the achieved goal. If the new planner $q'.\mathsf{Pl}$ provides a safe goal $g \in G$ for the new state, the goal gets updated to g. Otherwise, the goal gets updated to $\perp$ to indicate further updates for the planner are needed. These updates would be handled by the third type.

Third, re-plan transitions (repln) are enabled at a state q iff $q.G = \perp$. The environment and agent states do not change. The aim of this transition is to update the planner to provide a safe goal for the current state. Hence, the planner gets updated by Le. Then, its goal g for the current observed state is checked if safe by Sh. If safe, the goal gets updated to g. Otherwise, it

stays $\perp$ to allow further updates of the planner until one with a safe goal is reached.

3.5 Semantics and Safety

The semantics of our hybrid model is defined in the usual fashion [42, 43]. An *execution fragment* of $\mathcal{H}$ is an alternating sequence of trajectories and transition labels, $\alpha = \xi_0\ell_1\xi_1\ell_2\ldots\xi_k$, where each $\xi_i \in \mathcal{T}$, $\ell_i \in \{\mathsf{env}, \mathsf{pln}, \mathsf{repln}\}$ and $(\xi_i.lstate, \ell_{i+1}, \xi_{i+1}.fstate) \in \mathcal{D}$. The first state of an execution is defined as $\alpha.fstate = \xi_0.fstate$. An execution fragment α is an execution if it starts from the initial set, that is, $\alpha.fstate \in \Theta$. An execution fragment is *closed* if it is a finite sequence and all the trajectories have finite domain. Since we are interested in bounded time analysis, we only consider closed executions of $\mathcal{H}$ in this paper. The duration of a (closed) execution fragment α, denoted by $\alpha.dur$, is the sum of the duration ocf all its trajectories. The last state of a closed trajectory $\alpha = \xi_0\ell_1\ldots\xi_k$, is the last state of the last trajectory ξ_k in α, that is, $\alpha.lstate = \xi_k.lstate$. Notice that some trajectories may be of duration 0 or equivalently, a sequence of action can occur in 0 time. In fact, in the $\mathcal{H}$ automaton defined above, a pln action may be followed by a sequence of repln actions, in 0 time. Nevertheless, we can define $\alpha(t)$, that is, the state of $\mathcal{H}$ at time t, where $0 \leq t \leq \alpha.dur$ in the standard way as $\alpha'.lstate$ where α' is the longest prefix of α with duration t [43].

A state $q \in Q$ is *reachable* from a set of states $Q_0 \subseteq Q$ if there exists a closed execution fragment α such that $\alpha.fstate \in Q_0$ and $\alpha.lstate \in Q$. The set of all reachable states from Q_0 is denoted by $\mathsf{Reach}_{\mathcal{H}}(Q_0)$. For $\mathsf{Reach}_{\mathcal{H}}(\Theta)$ we simply write $\mathsf{Reach}_{\mathcal{H}}$ or Reach. A state q is *reachable within time* $T \geq 0$ from Q_0 if there exists a closed execution fragment α with $\alpha.dur \leq T$, $\alpha.fstate \in Q_0$, and $\alpha.lstate \in Q$. The bounded time reachable sets from Q_0 and Θ within time bound T, are defined analogously and are denoted by $\mathsf{Reach}_{\mathcal{H}}(Q_0, T)$ and $\mathsf{Reach}_{\mathcal{H}}(\Theta, T)$, or in the latter case simply by $\mathsf{Reach}(T)$.

Problem Statement. We say that $\mathcal{H}$ *achieves its mission* S_e^* if for each execution α there is a time t^* such that for all $t \geq t^*$, $\alpha(t).S_e \in S_e^*$. We say $\mathcal{H}$ is *safe* with respect to U if $(\mathsf{Reach}_{\mathcal{H}}(\Theta).S_a, \mathsf{Reach}_{\mathcal{H}}(\Theta).S_e) \cap U = \emptyset$.

Given all parameters of the hybrid system $\mathcal{H}$ except Sh *and* Le*, and given the requirements (S_e^*, U), our goal is to design a learner* Le *and a shield* Sh*, such that the resulting $\mathcal{H}$ is safe and achieves its mission.*

There are several other performance-related requirements we would like the Sh and Le to have[2] For example, Sh should be computationally effective, as it is called in run-time whenever a new goal is chosen by the planner, and Le should be able to give another planner that provides a safe goal if the previous planner goal was not safe. In other words, it should not have many repln transitions.

[2]The solution proposed in this paper does not address this wish list of performance requirements.

Chapter 4

FIRE-FIGHTING CASE STUDY

We consider the fire-fighting case study briefly mentioned in the introduction (Figure 4.1). The agent being controlled is a drone described by continuous-time dynamics. Its mission is to extinguish a forest fire that is sensed as a 2-dimensional grid-world environment. The drone has a water tank of limited capacity and therefore has to incrementally douse the fire by flying between a water source(s) and locations currently aflame, all the while avoiding obstacles scattered around in the environment.

We would like to use reinforcement learning for computing a high-level planner for the drone. We assume that the drone has a low-level controller that satisfies the tracking property (Definition 2), that is, it can effectively track the goals set by the planner within some tolerance.

4.1 Drone (Agent) Model

The drone is a quadcopter that has been outfitted with a water tank of limited capacity. Its flight dynamics adhere to those outlined in [15, 44] and are described below.

4.1.1 Drone dynamics model

The drone's state space is $S_a = \mathbb{R}^9 \times \{\mathsf{Emp}, \mathsf{Full}\}$. Let $r = [x, y, z]^T$ be the position of the center of mass in $\mathbb{R}^3$, $\dot{r}$ its velocity, $\ddot{r}$ its acceleration, m the mass, c the gravitational acceleration, $e_3 = [0, 0, 1]^T$, ϕ the roll angle, θ the pitch angle, ψ the yaw angle, $w \in \{\mathsf{Emp}, \mathsf{Full}\}$ the state of the water tank if its empty or full, γ the thrust input, and $\omega = [\omega_x, \omega_y, \omega_z]^T$ the body rotational rates control input. The drone state vector would be $s_a = (r, \dot{r}, \theta, \phi, \psi, w)$ and its control input would be (γ, ω). Its dynamics then follows the following

differential equation:

$$\ddot{r} = ce_3 + \frac{1}{m} J e_3 \gamma,$$

where J is the rotation matrix from the body frame to the world frame and is defined as follows:

$$J = \begin{bmatrix} c\theta c\psi & s\phi s\theta c\psi - c\phi s\psi & c\phi s\theta c\psi + s\phi s\psi \\ c\theta s\psi & s\phi s\theta s\psi + c\phi c\psi & c\phi s\theta s\psi - s\phi c\psi \\ -s\theta & s\phi c\theta & c\phi c\theta \end{bmatrix},$$

and $c\theta$ and $s\theta$ correspond to $\cos\theta$ and $\sin\theta$, respectively. Moreover,

$$\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} 1 & \sin(\phi)\tan(\theta) & \cos(\phi)\tan(\theta) \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \sin(\phi)\sec(\theta) & \cos(\phi)\sec(\theta) \end{bmatrix} \omega.$$

The control input is the sum of a feed-forward and feed-back control inputs described below.

4.1.2 Feed-forward controller

As stated in [15], the drone dynamics are differentially flat with flat output $\eta = [r^\intercal, \psi^\intercal]^\intercal$. That means that the state q and the control (γ, ω) can be represented as an algebraic function of $[\eta, \dot{\eta}, \ddot{\eta}, \dddot{\eta}]$. Given a three times differentiable desired trajectory $\eta_d = [r_d^\intercal, \psi_d^\intercal]^\intercal \in \mathcal{C}^3$ that we want the drone state to follow, one can derive the feed-forward controller to get:

$$\gamma_{ff} = -m \left\| \ddot{r}_d - ce_3 \right\|, \text{ and}$$

$$\begin{bmatrix} \omega_{1,ff} \\ \omega_{2,ff} \\ \omega_{3,ff} \end{bmatrix} = \begin{bmatrix} 1 & 0 & \sin(\theta_d) \\ 0 & \cos(\phi_d) & \sin(\phi_d)\cos(\theta_d) \\ 0 & -\sin(\phi_d) & \cos(\phi_d)\cos(\theta_d) \end{bmatrix},$$

where $\theta_d = \text{atan2}(\beta_a, \beta_b)$, $\phi_d = \text{atan2}(\beta_c, \sqrt{(\beta_a^2 + \beta_b^2)})$, $\beta_a = \ddot{x}_d \cos(\psi_d) - \ddot{y}_d \sin(\psi_d)$, $\beta_b = -\ddot{z}_d + c$, and $\beta_c = \ddot{x}_d \sin(\psi_d) + \ddot{y}_d \cos(\psi_d)$ [15]. The desired trajectory is computed based on the drone initial state $s_a \in S_a$ and its goal state $s'_a \in S_a$.

4.1.3 Feed-back controller

The feed-forward controller designed using differential flatness may not be sufficient as there may be modeling and tracking errors. These would be handled using a feed-back controller that aims to minimize the distance between the desired and actual trajectories. The components are as follows:

$$\gamma_{fb} = K_p\langle Je_3, r_d - r\rangle + K_d\langle Je_3, \dot{r}_d - \dot{r}\rangle, \text{ and}$$

$$\begin{bmatrix}\omega_{1,fb}\\ \omega_{2,fb}\\ \omega_{3,fb}\end{bmatrix} = K_p\begin{bmatrix}\phi_d - \phi\\ \theta_d - \theta\\ \psi_d - \psi\end{bmatrix} + K_d\begin{bmatrix}\dot{\phi}_d - \dot{\phi}\\ \dot{\theta}_d - \dot{\theta}\\ \dot{\psi}_d - \dot{\psi}\end{bmatrix} + \bar{K}_p\begin{bmatrix}y_d - y\\ x_d - x\\ 0\end{bmatrix},$$

The control input would be $[\gamma_{ff} + \gamma_{fb}, \omega_{ff} + \omega_{fb}]$.

In summary, once a goal state is chosen, a desired trajectory to get there from the current state is computed, and the dynamics become autonomous till reaching the goal state as the controller can be considered part of the dynamics. Finally, the water tank state w has zero dynamics.

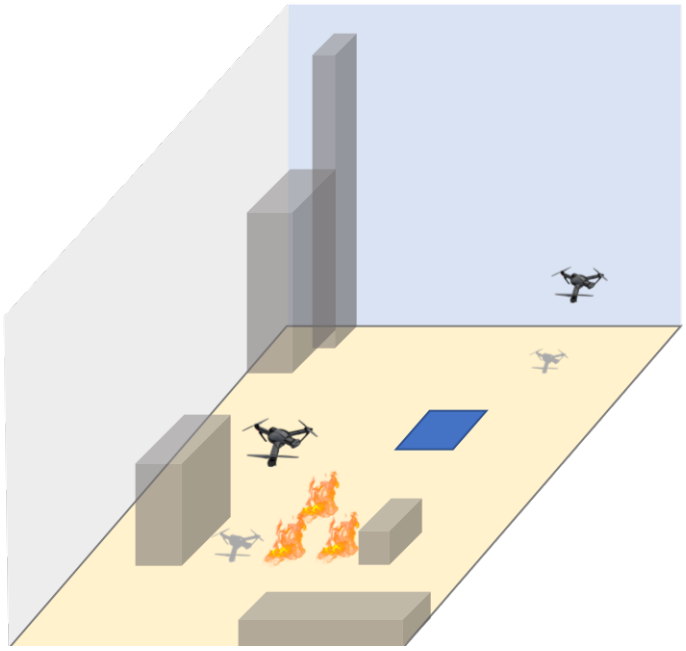

Figure 4.1: Illustration of fire-fighting scenario

4.1.4 Environment Model

The environment is modeled as a 2-dimensional grid with n cells over a bounded rectangle (see Figure 6.1). For a 4×4 environment $n = 16$. Each of the n cells can be in one of eight discrete states (as sensed by the drone, for instance through satellite imaging):

- FIRE{1,2,3,4} - currently on fire at one of four intensities
- UNLIT{5} - not currently on fire but may catch on fire
- EXTINGUISHED{6} - extinguished fire that cannot be ignited
- OBSTACLE{7} - any type of obstacle for the drone
- WATER{8} - source of water

Hence, the environment state space is $S_e = [8]^n$.

Now we describe the environment transition function En. Only a cell with a FIRE or UNLIT state would change state either because of fire propagation or because of water thrown by the drone. The fire propagates in the grid as a 2-dimensional convolution between a 2-dimensional kernel, called an *influence* matrix, and the environment. The influence matrix models an exogenous input like wind and is unknown to learner and the shield, but its effect on the state can be observed by the learner. More precisely, the environment update function En is defined as follows: At each cell in the environment, if it has a state of 5 or less, its updated state is the sum of the point-wise product between the *influence* matrix centered at it and the states of the cells of the environment while considering the state of any cell with a state larger than 4 as zero. Cells with states larger than 5 are kept unchanged.

In this thesis, we implemented our approach to only handle static obstacles; however, moving obstacles with (possibly nondeterministic) models could also be captured in En.

4.1.5 The System as a Hybrid Automaton

In this section, we will define all the sets and maps of the hybrid automaton $\mathcal{H}$ in Definition 1 for this case study except Sh and Le, which will be discussed separately in Chapter 5.

First, each grid cell of the environment is considered a goal. Hence, the goal set G is $[n]$. Moreover, the observation map $\mathsf{Obs} : S_a \to O_a := [n] \times \{\mathsf{Emp}, \mathsf{Full}\}$ maps any drone state $s_a \in S_a$ to the cell in the environment grid that the drone is currently in and its water tank state. The set of planners $\mathcal{P}$ we consider is the set of all functions (to be implemented as neural networks) that take as input the observed drone state $o_a \in O_a$ and the state of the environment $s_e \in S_e$ and output a cell in $[n]$ that the drone should go to. This set of planners ignores the current goal from the computation of a new one. We will further restrict this set to be networks with fixed architecture in Section 6.3. Overall, the state space of $\mathcal{H}$ is $Q = \mathbb{R}^9 \times \{\mathsf{Emp}, \mathsf{Full}\} \times [n] \times \{[n] \cup \perp\} \times [n] \times \mathcal{P}$.

The Grd is defined as follows: For each goal/cell $g \in G$, $\mathsf{Grd}(g)$ is an axis-parallel hyperrectangle in $\mathbb{R}^9 \times \{\mathsf{Emp}, \mathsf{Full}\}$ with the first two dimensions being a rectangle centered around the center of the corresponding cell.

The agent-environment joint update function AE is defined as follows: If the cell is a source of water, w gets mapped to Full while the environment state does not change. If the drone has $w = \mathsf{Full}$ and it resides in a cell with a state of FIRE of intensity larger than 1, w gets mapped to Emp and the cell fire intensity is decreased by 1. If instead the cell has a fire intensity of 1, it gets mapped to EXTINGUISHED state. For all other states, neither the drone nor the environment state changes. The initial set of states Θ is the set of all states with $s_a \in \cup_{g \in G} \mathsf{Grd}(g)$ and an environment state s_e with a single cell with a FIRE state and at least one source of water.

Chapter 5

FORSHIELD FRAMEWORK

In this chapter, we describe our approach FoRShield for implementing Sh and Le. There are two main challenges that an implementation of a Sh should tackle: (a) the agent has continuous-time possibly nonlinear dynamics as the drone of Section 4.1, and (b) many states of the agent map to the same observed state.

A shield Sh when called on an observed state $o \in O$ and a goal $g \in G$ should check if any execution fragment of $\mathcal{H}$ (continuous dynamics of the agent and the discrete dynamics of the environment) can lead to unsafe states (U), starting from any state in $\mathsf{Obs}^{-1}(o)$, and anytime before the fragment reaches $\mathsf{Grd}(g)$.

5.1 Algorithm for Shield

Let $Q_0 \subseteq Q$ and assume that all $q \in Q_0$ have $q.G = g$, for some $g \in G$. The set of reachable states by executions starting from Q_0 till the first planner transition, i.e., till all executions reach $\mathsf{Grd}(g)$, is denoted by $\mathsf{GoalReach}_{\mathcal{H}}(Q_0, g)$. If the set of agent states $Q_{a,0}$ in Q_0 is compact, there exists a $T > 0$ such that all executions starting from Q_0 would have reached their first pln transition. Then, $\mathsf{Reach}_{\mathcal{H}}(Q_0, T) \supseteq \mathsf{GoalReach}_{\mathcal{H}}(Q_0)$. Note that $\mathsf{GoalReach}_{\mathcal{H}}(Q_0)$ does not depend on the planner part of the states and all of its states have the same goal. Only the environment and agent states are the important information that are sought from computing such a set. Moreover, the dynamics of the agent and the environment are independent from each other in this set since they are only updated in a pln transition. Hence, one can compute the set of reachable states by the agent independently from those of the environment.

We introduce our approach that implements the Sh using reach set com-

putation. For any $o_a \in O_a$, $s_e \in S_e$, $g \in G$, and $p \in \mathsf{Pl}$, let Q_0 be the set of all states q with $q.S_a \in \mathsf{Obs}^{-1}(o_a)$, $q.S_e = S_e$, $q.G = g$, and arbitrary planners. To check the safety of having goal g at observed state (o_a, s_e), one would compute $\mathsf{GoalReach}_{\mathcal{H}}(Q_0)$. If it intersects U, it would return *unsafe* resembling the goal is unsafe. Otherwise, it would return *safe*. However, computing reach sets in all forms is computationally hard in general. Fortunately, there are several libraries and tools that can robustly over-approximate bounded time reach sets for nonlinear and hybrid models [45, 46, 47, 48, 49, 50, 51]. Sh would over-approximate computations of $\mathsf{GoalReach}_{\mathcal{H}}(Q_0)$ by computing over-approximations of bounded time reach sets using such tools.

Algorithm 1 Sh implementation

1: **input**: $o_a \in O_a, s_e \in S_e, g \in G$
2: Fix arbitrary $p \in \mathcal{P}$
3: $Q_0 \leftarrow \mathsf{Obs}^{-1}(o_a) \times \{s_e\} \times \{g\}, \{p\}$
4: $R \leftarrow$ Compute over-approximation of $\mathsf{GoalReach}_{\mathcal{H}}(Q_0)$
5: **if** $R \cap U = \emptyset$ **then**
6: return *safe*
7: **else**
8: return *unsafe*
9: **end if**

As stated in Theorem 1, it can be shown that the hybrid system with a Sh implemented as above is safe.

Theorem 1. *If* Sh *in* $\mathcal{H}$ *implemented as in Algorithm 1 and initially the goal is safe, that is, for all* $q \in \Theta$, $\mathsf{Sh}(\mathsf{Obs}(q.S_a), q.S_e, q.G) = \mathit{safe}$, *then the overall hybrid system* $\mathcal{H}$ *is safe (with respect to* U*).*

Proof. The bounded-time reachable set computation tool used in FoRShield is assumed to be sound. In other words, the set it computes should contain all states that can be reached in the bounded time. Using FoRShield, every goal executed by $\mathcal{H}$ is safe. FoRShield ensures that all possible executions given the current observed state will reach the Grd of the approved goal without intersecting with U. Consider any execution α of $\mathcal{H}$. There will be no state in the prefix of α before the first pln transition that intersects U by the assumption of the theorem that Sh returned *safe* and FoRShield over-approximates the reach set till reaching the first pln transition. By induction

on the pln transitions of α, consider any such transition and assume that there was no state reached by α before that belongs to U. Then, if Sh returned *safe* on the proposed goal by the new planner, there will be no state reached by α that belongs to U till the next pln transition by the assumption that the computed set by FoRShield solver over-approximates the reach set till the goal is achieved. If it returned *unsafe*, the goal would be set to $\perp$. Moreover, a sequence of repln transitions would occur till a planner that provides a goal that is checked by Sh to be safe is reached. During these transitions, the agent and environment state would not change. Hence, it will be safe by the induction assumption. Once the planner with the checked goal to be safe is reached, the execution will be safe till the next pln transition by, again, the fact that the computed set by FoRShield over-approximates the actual reach set. □

5.2 Faster Unsound Approaches

We introduce two alternative approaches that are computationally more efficient but less accurate than Algorithm 1. These alternatives will serve as baseline comparison points for our experiments.

- Sim. Given an observed agent state $o_a \in O_a$, environment state $s_e \in S_e$, and a goal $g \in G$, we choose an arbitrary $s_a \in \mathsf{Obs}^{-1}(o_a)$. Then, we forward simulate (Sim) the state using $\mathcal{E}$ for the environment state and by solving the O.D.E in Equation 3.1 till it reaches $\mathsf{Grd}(g)$ or U for the agent state. If U has been reached, Sh reports *unsafe*, otherwise, *safe*.

- Line. Given the current agent state s_a choose an arbitrary $s'_a \in \mathsf{Grd}(g)$ and connect s_a and s'_a by a straight line. The environment state is still evolved using $\mathcal{E}$. If the straight line along with the environment state intersects U, Sh would report *unsafe*, otherwise, *safe*.

Both approaches are not sound: they may report *safe* for unsafe executions and vice versa. However, they are usually much faster than computing reachsets.

5.3 Shield and Learner Implementation Pragmatics

FoRShield can be implemented in two ways. The first way aims to improve time efficiency at run time. FoRShield would do all the computations offline. Specifically, for any pair of goals $g, g' \in G$, an over-approximation of the GoalReach with initial set of the agent being $\mathsf{Grd}(g)$ and goal being g' is computed offline. Also, for each initial goal in Θ, an over-approximation for the reach set with initial set being the set of states in Θ with that goal is computed. All the results of the computations are cached. During run time, whenever Sh is called, FoRShield would propagate the dynamics of the environment using E and use the corresponding cached reach tube of the agent to check if they intersect U.

The second way is the default way. No computations happen offline. Instead, whenever Sh is called, FoRShield would compute the reach tube of the agent dynamics and propagate the environment dynamics using $\mathcal{E}$ and then checks the intersection with U.

Furthermore, for the Le end, we would design a reward function that maps the observed state and goal to a scalar. Then, we use a reinforcement learning algorithm to update the current planner based on the reported reward. In repln transitions, a $-\infty$ reward is reported so that ln would change the planner to not choose that goal again since it is unsafe.

Chapter 6

EXPERIMENTAL RESULTS AND IMPLEMENTATION DETAILS

In this chapter, we present the experimental results and discuss the implementation of FoRShield. Section 6.1 presents five different instances of the fire-fighting case study introduced in Chapter 4. Experimental results of the application of FoRShield to all the scenarios are discussed in Section 6.2. Lastly, the implementation for FoRShield is detailed in Section 6.3.

6.1 Scenarios

Each of the five scenarios tested against FoRShield is described below.

6.1.1 Scenario 1

In the scenario illustrated in Figure 6.1, the drone starts out in the upper left-hand corner of the 4×4 environment at $(0, 0)$. There is a water source located in the upper right-hand corner at $(0, 3)$. There is a single stationary fire of intensity 4 in the lower left-hand corner at $(3, 0)$. A single obstacle stands in the direct path between the water source and the fire at $(1, 2)$. To solve this scenario, the drone has to learn how to avoid the obstacle in at least one additional step.

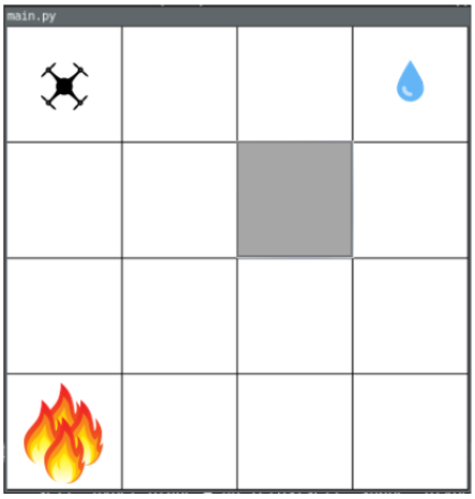

Figure 6.1: Scenario 1

6.1.2 Scenario 2

This scenario, illustrated in Figure 6.2, is the same as Scenario 1 but now the initial fire of intensity 4 at $(3, 0)$ spreads rightward across the bottom. The ideal solution to this scenario is to put out the fire that will spread to $(3, 1)$ to prevent the fire from spreading further and incrementally put out the original fire.

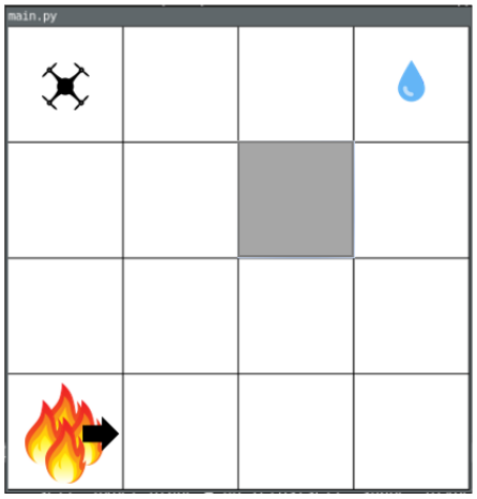

Figure 6.2: Scenario 2

6.1.3 Scenario 3

In the scenario illustrated in Figure 6.3, the drone starts out in the upper left-hand corner of the 4×4 environment at $(0, 0)$. There is a single water source located in the upper right-hand corner at $(0, 3)$. There is a single stationary fire of intensity 4 at $(3, 2)$ surrounded by multiple obstacles that block the direct path between the fire and the water. This scenario is similar to Scenario 1 but requires either 2 or 3 hops between the fire and the water depending on the dynamics of the agent.

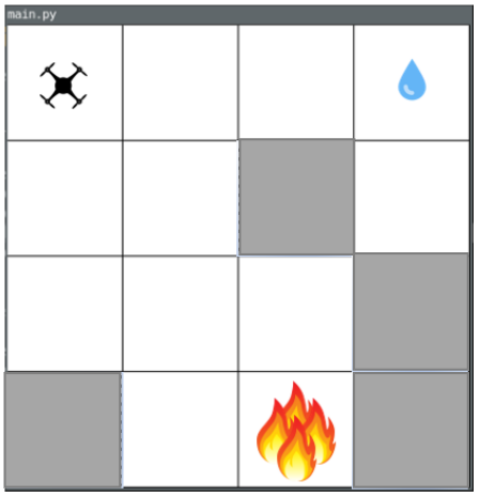

Figure 6.3: Scenario 3

6.1.4 Scenario 4

In the scenario illustrated in Figure 6.4, the drone starts out in the upper left-hand corner of the 4×4 environment at $(0, 0)$. There is a single water source located in the upper right-hand corner at $(0, 3)$. There is a fire of intensity 4 at $(3, 2)$ that spreads leftward across the bottom row. This scenario is a more complicated version of Scenario 2 because similar to Scenario 3, it requires extra actions by the agent each time it moves between the fire and water. The best solution to this scenario is for the drone to again put out the fire that spreads to $(3, 1)$ first to prevent the fire from spreading to $(3, 0)$ and then incrementally put out the original fire at $(3, 2)$.

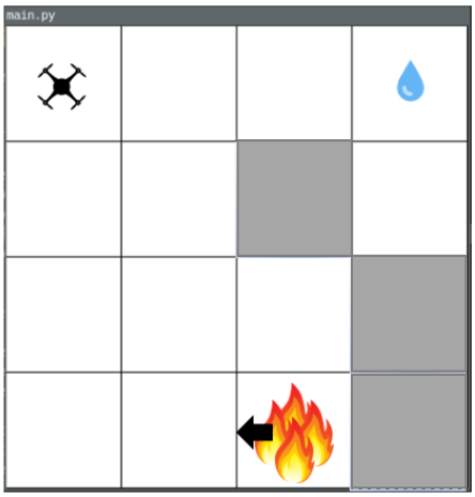

Figure 6.4: Scenario 4

6.1.5 Scenario 5

In the scenario illustrated in Figure 6.5, the drone starts out in the upper left-hand corner of the 4×4 environment at $(0, 0)$. There is a single water source located in the upper right-hand corner at $(0, 3)$. There are three obstacles that line the rightmost column underneath the water source. A fire of intensity 4 starts at $(1, 2)$ and spreads in two directions, downward and leftward. This is by far the most complicated scenario with the largest state space because the majority of cells may be affected by the fire spread. The optimal solution is to put out the spreading fire at $(1, 1)$ and $(2, 2)$ first to prevent the fire from spreading to any other cells and then incrementally put out the original fire.

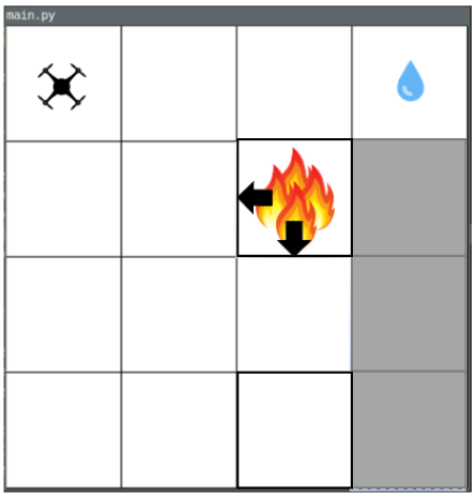

Figure 6.5: Scenario 5

6.2 Experimental Results

We evaluated the FoRShield approach presented in Chapters 3 and 5 against five instances of the fire-fighting case study described in Chapter 4. For each of the instances, we compare FoRShield with two other methods introduced in Section 5.2, to demonstrate the efficacy of FoRShield in controlling a drone safely around obstacles and achieving the mission target.

The data from these experiments are shown in Tables 6.1 and 6.2. All experiments were run on a laptop equipped with an Intel Core i5-6200U CPU and 8 GBs of RAM. The key observations from these experiments are as follows.

Table 6.1: Run-time results across different scenarios using FoRShield (FSh), line simulation (Line), and forward simulation (Sim)

Experiments						
Scenarios	Verification time (hh:mm:ss)			Training time (hh:mm:ss)		
	Line	Sim	FSh	Line	Sim	FSh
Scenario 1	00:07:57	00:05:35	00:21:40	01:16:16	00:54:43	02:09:41
Scenario 2	02:36:58	09:56:31	00:21:40	15:42:58	15:29:51	05:05:29
Scenario 3	01:45:39	01:42:50	00:21:16	02:14:13	02:03:22	08:54:06
Scenario 4	03:40:38	03:02:17	00:07:16	07:32:20	04:13:00	04:42:26
Scenario 5	-	-	00:12:44	-	-	187:02:49

Table 6.2: Trajectory and verification results across different scenarios using FoRShield (FSh), line simulation (Line), and forward simulation (Sim)

Experiments									
Scenarios	Total verifications			Unsafe verifications			Unsafe trajectories		
	Line	Sim	FSh	Line	Sim	FSh	Line	Sim	FSh
Scenario 1	26431	18809	256	4334	2851	132	34	20	0
Scenario 2	575934	487482	256*	52878	9301	132*	3560	361	0
Scenario 3	68929	56071	256	14343	10452	195	2792	1136	0
Scenario 4	244611	136335	256	15682	11080	156	4	397	0
Scenario 5	-	-	256	-	-	109	-	-	0

Learned planners are near-optimal. Across all the scenarios except for the last one, FoRShield learned near-optimal planners if not an optimal planner, completing the mission target of putting out the fire in the shortest time possible. In fact, both the unsound strategies, Line and Sim, also found near optimal planners. The scenarios used in these experiments are small enough that the optimality of the strategies can be derived empirically, but these results will carry-over to more complicated scenarios as well. In order for the RL algorithm to maximize the total reward per episode, it inherently needs to converge to a policy with minimal time.

The introduction of a fire propagating in two directions in Scenario 5 greatly increased the state space of the problem. In Scenario 1, the complete state space is $5 \times 4^2 \times 2 = 160$ where 5 is the number of states the burning cell may take, 4^2 for the current location of the agent in the environment, and 2 to indicate whether the agent has any water. In contrast, in Scenario 5, the state space blows up to slightly less than $5^9 \times 4^2 \times 2 = 62,500,000$ due to the increase in cells that may catch fire. Due to the predefined episode limit, the agent was unable to learn a viable policy to extinguish a fire; however, if the agent was allowed to train sufficiently longer, it would have definitely discovered an optimal policy as indicated by the increase in rewards over the training period.

Execution of Line against Scenario 5 was attempted but the learning process eventually halted as it became stuck in particular episode and did not make any progress after one day. Due to this issue, Sim was not run against Scenario 5. Even though neither of these strategies were tested, it can be expected that similar results would be achieved as prior scenarios.

Safe RL with FoRShield is feasible. FoRShield always performed the smallest number of verifications and encountered the minimal number of unsafe verifications because it cached its reach-set computations, whereas Line and Sim are necessarily online methods and verified every action the agent performed. Additionally, as expected, FoRShield completely eliminated the risk of the agent ever performing any unsafe actions during training while this was not the case for Line and Sim where multiple unsafe actions were taken.

FoRShield's total running time decreases with scenario complexity. The verification running time of FoRShield remains mostly constant across all the scenarios. The decrease in verification time for later scenarios can be attributed to an increase in unsafe cells which reduces the number of times refinement computations are necessary. Compared to the Line and Sim strategies, FoRShield tends to be slower for simpler scenarios where verifying the environment takes longer than just running verifications online with each action execution, but this trend reverses for more complex examples where many more actions are performed. By caching its verification results, FoRShield just does simple look-ups to check whether a particular action is safe to execute.

In general, it is difficult to make any comparisons in training time across the three strategies because the time is highly correlated with initialization of the networks which is random. This may explain the enormous differences in training times particularly in Scenarios 2 and 3. In order to make a fair comparison, it would be necessary to rerun all the strategies for each of the Scenarios multiple times.

6.3 Implementation Details

The FoRShield framework—the shield, environment, agent, and learner—is implemented in Python and uses a variety of modules such as Numpy, TensorFlow, and OpenAI Gym to name a few. FoRShield and the Sim method both utilize DryVR [52] as a verification back-end to simulate the continuous dynamics of the drone model while Line uses a simple intersection algorithm. The fire-fighting environment and learner were both derived from existing

implementations and modified accordingly for the specific requirements of the framework.

For all our experiments, the learner is an instance of an actor-critic algorithm. Our implementation of the algorithm uses two separate networks, one each for the actor and the critic. Both the actor and critic networks are described in more detail below.

The actor network is a 4-layered fully connected neural network with 256 nodes in the first hidden layer, 128 nodes in the second hidden layer, 64 nodes in the third hidden layer and a variable number of nodes in the output depending on the dimensions of the environment. Each of the hidden layers is equipped with the ReLu function and the output layer utilizes the softmax function. Given an environment of size $(r \times c)$, the output layer has a total of $r + c$ nodes of which r nodes correspond to selecting a particular row and c nodes correspond to selecting a particular column. Each set of r and n nodes utilizes its own softmax function to compute a particular coordinate destination that the agent should navigate to.

The critic network is a 3-layered fully connected neural network with 128 nodes in the first hidden layer, 64 nodes in the second hidden layer, and a single node in the output layer. Each of the hidden layers is equipped with the ReLu function, and the output layer uses a plain linear function.

To create a modular system, the framework separates each of the major components into individual modules that are linked together in the experiment section.

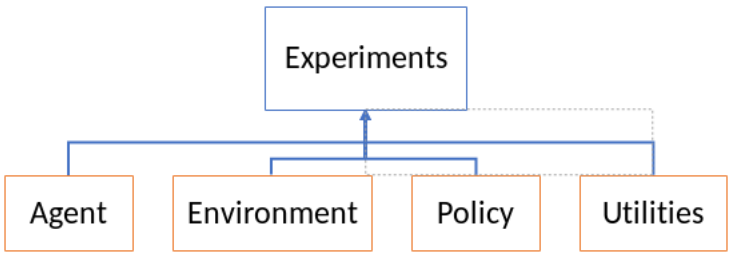

Figure 6.6: Modular software architecture

Similar to the pre- and post- placements of the shield in [34], the verification of the environment can happen in three distinct locations: before (pre-), during (peri-), or after (post-) the training of the fire-fighting policy by the agent. In the above experiments, the FSh approach utilized the

pre-verification algorithm while Line and Sim utilized the peri-verification algorithm. Following are a brief explanation and diagram for each algorithm.

6.3.1 Pre-verification Algorithm

In this setup shown in Figure 6.7, the environment is verified beforehand and the results are cached into either a data-structure or file. During training, only a simple lookup is required to get the results.

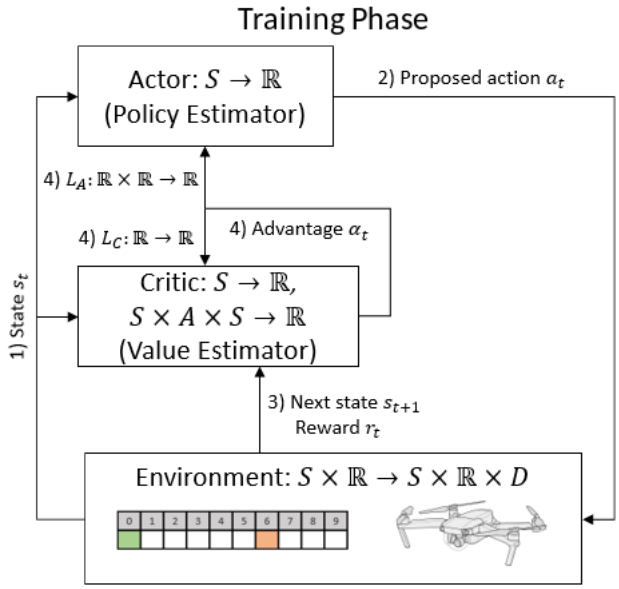

Figure 6.7: Diagram of the pre-verification algorithm

Suppose that at time t, the algorithm has just completed a single iteration of training and is about to begin the next one. The training loop then runs as follows:

1. The state at time t, s_t, is fed into the actor network.
2. The actor network processes s_t and proposed an action a_t that is sent to the environment.
3. The environment checks whether the proposed action is safe against cached verification results. If the action is safe, it is executed and the environment is updated accordingly; otherwise no action is taken and a large negative reward is output. In either case, the environment returns the next state s_{t+1} and the reward r_t to the critic.

4. The critic computes an advantage a_t using s_t, s_{t+1}, and r_t. This advantage is used to compute the loss function to update both the actor and critic networks.

6.3.2 Peri-verification Algorithm

In this setup shown in Figure 6.8, the environment is verified online during training. Each time the agent proposes an action, the action will first be verified to check its safety.

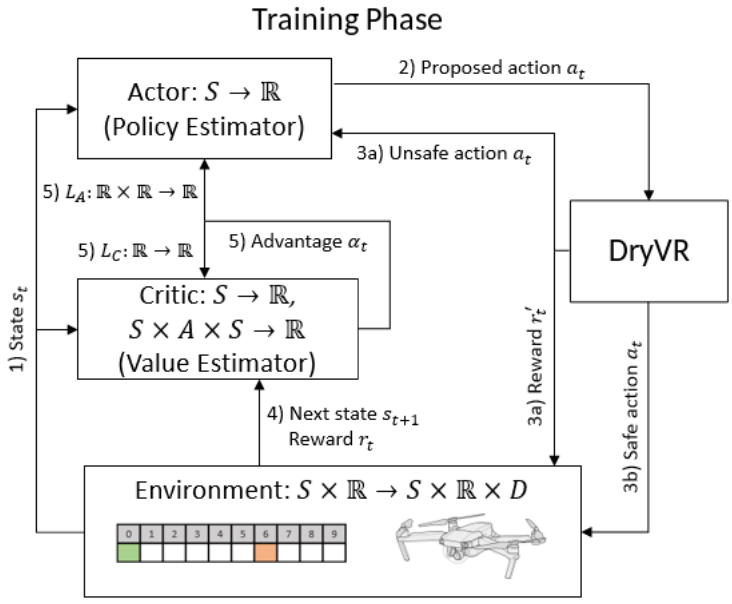

Figure 6.8: Diagram of peri-verification algorithm

Suppose that at time t, the algorithm has just completed a single iteration of training and is about to begin the next one. The training loop then runs as follows:

1. The state at time t, s_t, is fed into the actor network.

2. The actor network processes s_t and proposes an action a_t that is sent to DryVR to verify whether the action is safe.

3a. If DryVR verifies the proposed action to be unsafe, a large negative reward is passed to the environment.

3b. If DryVR verifies the proposed action to be safe, the action passes through to the environment.

4. In either of the previous cases, the environment returns the next state s_{t+1} and the reward r_t to the critic.

5. The critic computes an advantage a_t using s_t, s_{t+1}, and r_t. This advantage is used to compute the loss function to update both the actor and critic networks.

6.3.3 Post-verification Algorithm

In this setup shown in Figure 6.9, a policy is first learned by the agent. Once the full training phase has completed, the resultant policy is verified to check whether it violates any safety constraints. If any action is unsafe, that action is marked as unsafe and the agent will relearn a completely new policy.

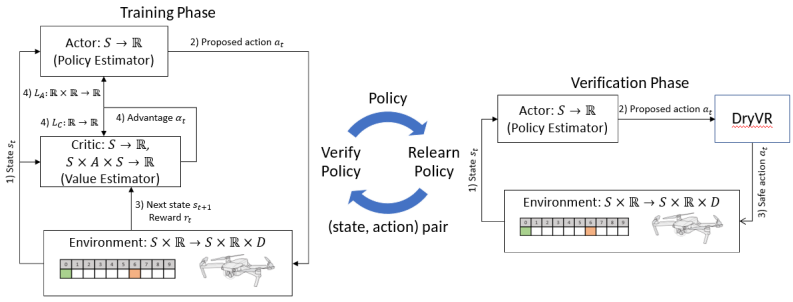

Figure 6.9: Diagram of post-verification algorithm

Suppose that at time t, the algorithm has just finished verifying a policy and determined that the policy is unsafe.

1. A policy is learned in the training phase similarly as in pre-verification but the only cached verification results are the concatenations of the previous results from the verification phase.

2. Once a policy has been learned in the training phase, it is passed to the verification phase to check whether any of the actions in the policy violate any safety constraints. If so, then violating action will be recorded in the cached results and a new policy will be learned with new addition in mind; otherwise, the algorithm will terminate and output the learned policy.

Chapter 7

CONCLUSION AND FUTURE WORK

As machine learning becomes an integral part of CPS products, it will become necessary to develop methods for specifying hard safety constraints that these systems must not violate and for verifying their correctness. Researchers have started focusing their attention on this issues as outlined in the first chapter.

In this thesis, we introduced FoRShield, a step toward realizing such methods. FoRShield is a framework for learning safe and near-optimal policies, particularly for agents with nonlinear continuous dynamics. We developed a formal specification of this framework and illustrated its use in a fire-fighting case study. Our experiments demonstrate that FoRShield is a viable approach capable of functioning in environments with large state spaces. We have preliminary results showing that FoRShield can discover nontrivial strategies for putting out a moving fire around obstacles.

There are multiple directions to explore within the FoRShield framework. The first is to improve the learning algorithm to be able to handle more complex environments. Although the current implementation is able to handle relatively complex environments as demonstrated in the experiments, it takes a long time to learn optimal policies. The second direction is to increase the complexity of the discrete environment dynamics to include events such as the depletion of water source(s) and movement of obstacles. Along these lines, the formal framework can be extended to handle environments that evolve according to continuous dynamics rather than just the discrete ones. Such a modeling scheme would allow us to capture more real-world scenarios. A third direction is to study strategies for coordinating multiple agents toward a shared mission target. Developing safe coordination techniques would be applicable to examples such as intersection crossings of autonomous vehicles with minimal wait time.

A closely related research direction that is worth exploring is multi-objective reinforcement learning (MoRL). In MoRL, agents attempt to learn policies

that necessarily optimize for multiple, possibly contradicting, objectives. Depending on the problem, safe RL can be framed as a MoRL problem, where the agent is trying to maximize its safety (or alternatively, minimize its risk) while simultaneously maximizing some reward. Current MoRL theories cannot handle objectives that are non-linearly related and can only handle linear combinations of multiple objectives. Any developments in this area would be a boon for RL, particularly safe RL.

REFERENCES

[1] D. Wakabayashi, "Self-driving uber car kills pedestrian in arizona, where robots roam," https://www.nytimes.com/2018/03/19/technology/uber-driverless-fatality.html.

[2] "Tesla car that crashed and killed driver was running on autopilot, firm says," https://www.theguardian.com/technology/2018/mar/31/tesla-car-crash-autopilot-mountain-view.

[3] "Uber halts self-driving car tests after death," https://www.bbc.com/news/business-43459156.

[4] "Waymo's self-driving cars have traveled 8 million miles on public roads," https://fortune.com/2018/07/20/waymo-self-driving-cars-8-million-miles/.

[5] "Tesla reaches 10 billion electric miles with a global fleet of half a million cars," https://electrek.co/2018/11/16/tesla-fleet-10-billion-electric-miles/.

[6] http://www.businessdictionary.com/definition/Goodharts-law.html.

[7] https://righthook.io/.

[8] E. W. Dijkstra, "On the reliability of programs," https://www.cs.utexas.edu/users/EWD/transcriptions/EWD03xx/EWD303.html, 2005.

[9] "Full self-driving hardware on all cars," https://www.tesla.com/autopilot, 2018.

[10] A. Kurakin, I. Goodfellow, and S. Bengio, "Adversarial examples in the physical world," *arXiv preprint arXiv:1607.02533*, 2016.

[11] T. Dreossi, S. Jha, and S. A. Seshia, "Semantic adversarial deep learning," *arXiv preprint arXiv:1804.07045*, 2018.

[12] S. Jha, S. S. Banerjee, J. Cyriac, Z. T. Kalbarczyk, and R. K. Iyer, "Avfi: Fault injection for autonomous vehicles," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 2018, pp. 55–56.

[13] A. H. M. Rubaiyat, Y. Qin, and H. Alemzadeh, "Experimental resilience assessment of an open-source driving agent," *arXiv preprint arXiv:1807.06172*, 2018.

[14] "Ghostriding for the masses." https://comma.ai/.

[15] L. Wang, E. A. Theodorou, and M. Egerstedt, "Safe learning of quadrotor dynamics using barrier certificates," *arXiv preprint arXiv:1710.05472*, 2017.

[16] V. A. Papavassiliou and S. Russell, "Convergence of reinforcement learning with general function approximators," in *IJCAI*, 1999, pp. 748–757.

[17] R. Hecht-Nielsen, "Theory of the backpropagation neural network," in *Neural Networks for Perception*. Elsevier, 1992, pp. 65–93.

[18] J. Garcıa and F. Fernández, "A comprehensive survey on safe reinforcement learning," *Journal of Machine Learning Research*, vol. 16, no. 1, pp. 1437–1480, 2015.

[19] S. Hochreiter, "Untersuchungen zu dynamischen neuronalen netzen," *Diploma, Technische Universität München*, vol. 91, no. 1, 1991.

[20] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, 2010, pp. 249–256.

[21] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, 2010, pp. 807–814.

[22] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*. Ieee, 2009, pp. 248–255.

[23] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, 2012, pp. 1097–1105.

[24] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[25] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Computer Vision and Pattern Recognition (CVPR)*, 2015.

[26] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 770–778.

[27] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski et al., "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, p. 529, 2015.

[28] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot et al., "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, p. 484, 2016.

[29] OpenAI, "Openai five," https://blog.openai.com/openai-five/, 2018.

[30] O. Vinyals, T. Ewalds, S. Bartunov, P. Georgiev, A. S. Vezhnevets, M. Yeo, A. Makhzani, H. Küttler, J. Agapiou, J. Schrittwieser et al., "Starcraft ii: A new challenge for reinforcement learning," *arXiv preprint arXiv:1708.04782*, 2017.

[31] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," *arXiv preprint arXiv:1412.6572*, 2014.

[32] P. Van Wesel and A. E. Goodloe, "Challenges in the verification of reinforcement learning algorithms," 2017, NASA Langley Research Center.

[33] W. Xiang, P. Musau, A. A. Wild, D. M. Lopez, N. Hamilton, X. Yang, J. Rosenfeld, and T. T. Johnson, "Verification for machine learning, autonomy, and neural networks survey," *arXiv preprint arXiv:1810.01989*, 2018.

[34] M. Alshiekh, R. Bloem, R. Ehlers, B. Könighofer, S. Niekum, and U. Topcu, "Safe reinforcement learning via shielding," *arXiv preprint arXiv:1708.08611*, 2017.

[35] S. Junges, N. Jansen, C. Dehnert, U. Topcu, and J.-P. Katoen, "Safety-constrained reinforcement learning for MDPs," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2016, pp. 130–146.

[36] N. Jansen, B. Könighofer, S. Junges, and R. Bloem, "Shielded decision-making in MDPs," *arXiv preprint arXiv:1807.06096*, 2018.

[37] O. Bastani, Y. Pu, and A. Solar-Lezama, "Verifiable reinforcement learning via policy extraction," *arXiv preprint arXiv:1805.08328*, 2018.

[38] S. Ross, G. Gordon, and D. Bagnell, "A reduction of imitation learning and structured prediction to no-regret online learning," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, 2011, pp. 627–635.

[39] A. Verma, V. Murali, R. Singh, P. Kohli, and S. Chaudhuri, "Programmatically interpretable reinforcement learning," *arXiv preprint arXiv:1804.02477*, 2018.

[40] S. Bhupatiraju, K. K. Agrawal, and R. Singh, "Towards mixed optimization for reinforcement learning with program synthesis," *arXiv preprint arXiv:1807.00403*, 2018.

[41] B. Gebremichael and F. Vaandrager, "Specifying urgency in timed I/O automata," in *Software Engineering and Formal Methods, 2005. SEFM 2005. Third IEEE International Conference on.* IEEE, 2005, pp. 64–73.

[42] D. K. Kaynar, N. Lynch, R. Segala, and F. Vaandrager, "The theory of timed I/O automata," *Synthesis Lectures on Distributed Computing Theory*, vol. 1, no. 1, pp. 1–137, 2010.

[43] S. Mitra, "A verification framework for hybrid systems," Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA 02139, September 2007.

[44] D. Zhou and M. Schwager, "Vector field following for quadrotors using differential flatness," *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 6567–6572, 2014.

[45] X. Chen, E. Ábrahám, and S. Sankaranarayanan, "Flow*: An analyzer for non-linear hybrid systems," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, N. Sharygina and H. Veith, Eds. Springer Berlin Heidelberg, 2013, vol. 8044, pp. 258–263.

[46] G. Frehse, C. L. Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler, "Spaceex: Scalable verification of hybrid systems," in *CAV*, 2011, pp. 379–395.

[47] C. Fan, B. Qi, S. Mitra, and M. Viswanathan, "DryVR: Data-driven verification and compositional reasoning for automotive systems," in *International Conference on Computer Aided Verification.* Springer, 2017, pp. 441–461.

[48] P. S. Duggirala, S. Mitra, M. Viswanathan, and M. Potok, "C2e2: a verification tool for stateflow models," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 2015, pp. 68–82.

[49] A. Donzé, "Breach, a toolbox for verification and parameter synthesis of hybrid systems," in *C.A.V.*, 2010.

[50] E. Asarin, O. Bournez, T. Dang, and O. Maler, "Approximate reachability analysis of piecewise-linear dynamical systems," in *Hybrid Systems: Computation and Control*, ser. LNCS, B. Krogh and N. Lynch, Eds., vol. 1790. Hybrid Systems: Computation and Control, 2000, pp. 20–31.

[51] S. Kong, S. Gao, W. Chen, and E. M. Clarke, "dReach: δ-reachability analysis for hybrid systems," in *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015*, 2015, pp. 200–205.

[52] C. Fan, B. Qi, and S. Mitra, "Data-driven formal reasoning and their applications in safety analysis of vehicle autonomy features," *IEEE Design & Test*, vol. 35, no. 3, pp. 31–38, 2018.